(2)

WORKING MATERIAL

FOR THE LECTURES OF

A. NERODE

LECTURES ON INTUITIONISTIC LOGIC: PART II

INTERNATIONAL SUMMER SCHOOL

ON

LOGIC, ALGEBRA AND COMPUTATION

MARKTOBERDORF, GERMANY, JULY 25 - AUGUST 6, 1989

THIS SUMMER SCHOOL IS ORGANIZED UNDER THE AUSPICES OF THE TECHNISCHE UNIVERSITÄT MÜNCHEN AND IS SPONSORED BY THE NATO SCIENCE COMMITTEE AS PART OF THE 1989 ADVANCED STUDY INSTITUTES PROGRAMME. PARTIAL SUPPORT FOR THE CONFERENCE WAS PROVIDED BY THE EUROPEAN RESEARCH OFFICE AND THE NATIONAL SCIENCE FOUNDATION AND BY VARIOUS INDUSTRIAL COMPANIES.

LECTURES ON INTUITIONISTIC LOGIC: PART II

A. Nerode*

Mathematical Sciences Institute

Cornell University

Ithaca, New York 14853

Abstract

Intuitionistic logic has become ubiquitous in computer science. An important question is how to teach intuitionistic logic to computer science and mathematics students. We attempted to begin such an exposition in Nerode [1989] using Kripke models which allow us to represent "states of knowledge" about machines. The present notes are an introduction, without proofs, to the semantics of recursive realizability and the Curry–Howard isomorphism. This is the subject behind term extraction functional computer languages such as ML or NuPRL.

§1. Orientation. Here is a paradigm.

Deductions are computations,

Computations are deductions.

We do not intend this in a narrow sense. Here are some generic examples.

a) A first example of "deduction as computation" is backward chaining. An algorithm for searching for a deduction of A from $B_1, \ldots, B_n$ may start with a desired

conclusion A, and then apply all rules of deduction in reverse systematically at intermediate stages to see how each statement could arise as an intermediate conclusion from intermediate premises. The algorithm then terminates when A has been traced back through various stages to the assumptions $B_1, ..., B_n$, and does not terminate if it never succeeds in constructing a deduction of A from $B_1, ..., B_n$. Look at any treatment of logic resolution theorem provers (Robinson [1965]), or an ideal PROLOG interpreter or compiler (Lloyd [1989]).

b) A second example of "deduction as computation" is forward chaining. An algorithm for searching for a deduction of A from $B_1, ..., B_n$ may start with $B_1, ..., B_n$ as data, applying all rules of deduction systematically at intermediate stages to intermediate premises to obtain intermediate conclusions. The program then terminates when A is obtained as a conclusion, and therefore when a deduction of A from $B_1, ..., B_n$ has been obtained An idealization of the classificatory expert system language OPS5 is an example (Brownston et al. [1985]).

In both of these examples Ia and Ib all the deduction rules correspond to internal commands of the appropriate computer language, which may be sequential or parallel. What is implemented in a compiler or interpreter is a systematic search procedure for such a deduction, a proof procedure which is complete in the sense of a completeness theorem for the underlying logic. (We ignore the short–cuts taken by language writers to get efficiency by violating correctness and completeness.)

c) A third example of "deduction as computation" is the implementation of intuitionistic deduction. Intuitionistic deduction is the form of deduction in systems based on Heyting's work of the 1930's, which restricts the use of logical deduction rules to those used in the constructive reasoning of L. E. J. Brouwer, the distinguished Dutch mathematician who originated Intuitionism after the turn of the century. One can search for a deduction as above. But often, unlike the cases above, the emphasis is not on searching for a deduction, but rather on converting a given intuitionistic deduction which proves the existence of an object, into an algorithm for computing that object. Intuitionistic deductions are constructive. One can write a program which, applied to any input which is a constructive proof that an object exists, terminates with an output which is a program for computing that object. This is called

"extracting programs from (constructive) proofs".

Conceptually this idea of extraction goes back to Heyting [1934] and Kleene [1945]. When applied to a constructive proof that a definition defines a function F on some data types (the function is an object, in the terminology above), this extraction program produces a program for computing the function F. This is the basis for many recent high level functional programming languages. These include AUTOMATH (DeBruijn [1973, 1980]), NuPRL (Constable [1986], Martin–Löf [1984, 1987])), ML and other polymorphic languages (Reynolds [1974], Girard [1972, 1986]) such as the theory of "Constructions" of Huet and Coquand [1985]. Extending this relation between intuitionistic deduction and computation to wider domains of computer science problems is an important area of current research. One of the main reasons for studying intuitionistic systems in computer science is that among its likely future achievements is the production of guaranteed correct programming languages.

d) Here is an example of "computation as deduction". Think of a computation as proceeding in stages, and as having a unique state at each stage and a unique input history. Allow parallelism and non–determinism. Each successive state is one of the possible successors of the previous state of the processor and input history. The possible finite sequences of states compatible with machine operation rules are the possible finite execution sequences. A sequential machine is one with only one execution sequence. A crude translation from execution sequences to deductions is this. Given the input history H and the current state S and one possible successor state S' (there may be many), associate one rule of inference $I_{H,S,S'}$ with H , S as premises, and S' as conclusion. Finite deductions from an initial state and history represent execution sequences. We are deducing possible execution sequences. This is an operational point of view, and uses states. There are many other points of view on computation, and each admits a corresponding deduction system so far as we can see. From another point of view, each execution sequence is a model of a theory, rather than a deduction *in* the theory.

e) Every system in which program correctness (i.e., that programs meet their program specification) can be proved can be construed as a logical system, for in what else can one prove anything? There is an enormous quantity of ongoing work concerning sequential programs on program specification, program development, and program correctness. When substantial code is involved, carrying out these tasks for sequential

programs is itself an art of cooperation of many hands, fraught with difficulties in maintaining accuracy and efficiency, and generally not carried out in a formal way in practice. Making as much of the process as routine as possible, and then automating these routine parts by programs themselves is commercially desirable and theoretically very interesting. It is not at all clear *a priori* what methodologies can be developed, but intellectual tools are slowly emerging, many from mathematical logic. Increasingly these tools come from, or are simply related to, themes of intuitionistic logic rather than themes of classical logic. This is because intuitionistic systems permit extraction of programs from deductions. Computer aided extraction of programs from program specifications is an ultimate aim. The validation of the computer aided extraction would then automatically show the correctness of all programs extracted.

f) Here is another role of the theory of deductions, traditionally called proof theory. Proof theory deals with proof–simplification, originally intended for applications to consistency proofs. Proof–theoretic analyses often yield algorithms for converting deductions into deductions of a very restricted form. To search for a deduction then becomes a search for one of a restricted form. This often forms the basis for new forward or backward chaining search–for–a–deduction compilers and interpreters. Such a new interpreter or compiler may run more efficiently than those based on older procedures since what has to be searched for comes out of a smaller class. The evolution of backward chaining resolution theorem proving bears this out, starting with simple resolution on ground terms, proceeding to unification, and then going on to SLD resolution, etc. (Robinson [1965]). The same holds of algorithmic improvements in the deduction systems for forward chaining systems. As for the intuitionistic deduction systems mentioned above, they depend exactly on the proof–simplification procedures of proof theory. In fact, they compute by simplifying deductions, and terminate when the simplification is complete (they terminate in a normal term or deduction, see below).

Corresponding work for extraction of code for distributed and concurrent programs from proofs is in its infancy. We do not know enough about how to proceed from program specification to code dividing up execution among many processors in such a way as to avoid sequential bottlenecks, if this is indeed possible at all.

In trying to understand any new quantitative subject matter, any kind of rigorous reasoning is, of course, allowed. The coarsest reasoning is classical reasoning in which proofs of the existence of objects may even be obtained by supposing the object does not

exist and then getting a contradiction. A typical example of such a situation in computer science is giving a non–constructive proof that in a computing environment with several computing agents which compete for use of a resource (such as disk access), the program governing use of the resource has the property that for each agent there exists a time at which that agent will be given access (but we may not know how much delay each computing agent will encounter in waiting for access to the resource).

But usually only "constructive" reasoning is likely to be of practical use. One actually has to get one's hands on algorithms for computing things asserted to exist and bounds on their running time to make any use of them. In the above instance, knowing that disk access will eventually take place is of little use without a bound on how long that may take, and this time better be reasonable, or the program will not be used.

Can a given instance of non–constructive reasoning be replaced by constructive reasoning? This is often a highly non–trivial mathematical question, only answerable by hard mathematical work. See e.g, Friedman [1978]. Often a non–constructive existence proof is "cheap" in terms of the time necessary to establish it, and is also quite general and aesthetically pleasing. Finding a constructive version of that proof requires looking at exactly how existence was obtained, and filling in whatever additional information and algorithms are needed in order to compute the thing asserted to exist.
First, we give a standard formalization of intuitionistic predicate logic by natural deduction in the form due to Prawitz.

§2. Intuitionistic natural deduction. In the early 1930's Heyting developed formal intuitionistic predicate logic to describe Brouwer's mode of constructive reasoning. We use the Prawitz [1965] formulation of intuitionistic deductions as trees formed using Gentzen's introduction and elimination rules. We take for granted the usual inductive definition of predicate logic formulas based on "$\wedge$" (and), "$\vee$" (or), "$\neg$" (not), "$\exists$" (there exists), "$\forall$" (for all), "F" (falsehood), together with relation symbols and variables. We do not introduce a "T". In formulas we omit as many brackets as possible, and sometimes use both round and square brackets in the same formula for legibility.

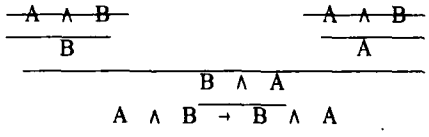
Here is an example of an ordinary mathematical proof and its natural deduction equivalent. This will help in puzzling out the formal definition below.

Example. Consider how one ordinarily proves $A \wedge B \rightarrow B \wedge A$. Explanation of the rules

used is on the next page.

ORDINARY PROOF	REASON
Suppose $A \wedge B$.	assumption
Then $A \wedge B$ yields A.	$\wedge$–elimination
Also $A \wedge B$ yields B.	$\wedge$–elimination
Then B, A yield $B \wedge A$.	$\wedge$–introduction
Conclude $A \wedge B \rightarrow B \wedge A$.	$\rightarrow$–introduction
	cancellation of assumption

This yields the natural deduction proof



Cancellation takes place when we legally drop an hypothesis as the conclusion is reached. For example, if A is an hypothesis and B is the conclusion, and we want to conclude $A \rightarrow B$, then we can cancel A since it is not needed to justify the implication. All deductions will be finite labelled trees, with the root at the bottom and the leaves (upper most nodes farthest from the root) at the top. The labels are on the nodes. The label on the root node is the conclusion. In the case of a non–leaf node, the label on the node is a formula. In the case of leaf nodes, every label is a formula or a canceled formula (a formula with a horizontal line through the middle). The uncanceled formulas on the leaves are called the assumptions of the deduction, the canceled formulas on the leaf nodes are called the canceled assumptions. Individual occurrences of assumptions will be distinguished at all times. In particular, there will be a process of "canceling" assumptions (putting a horizontal bar through the assumption), and we are allowed to cancel none, some, or all occurrences of a given assumption.

All natural deduction rules are to be regarded as constructing new finite labelled trees from old finite labelled trees, possibly canceling some assumptions in the process, and always preserving previous cancellations. Among the operations listed below, "implication introduction", "or elimination", and "existential quantifier elimination" are the only ones which newly cancel assumptions, the rest do not alter the canceled or uncanceled character of the assumptions. When we have no need to refer to assumptions we write the tree as

$$\vdots$$
$$B$$

If we need to refer to a formula A as an assumption (which may occur vacuously on the tree), we write

$$\begin{array}{c} A \\ \vdots \\ B \end{array}$$

We need to start with atomic deductions. These are of the form

$$A$$

where the assumption and the conclusion are exactly the same.

In the deduction rules we separate the conclusion from the premises by a horizontal bar. This is not present in the formal definition of a deduction as a labelled tree, but is the standard way of writing natural deductions with paper and pencil.

$\wedge$ – elimination.
From a deduction of $A \wedge B$

$$\begin{array}{c} \vdots \\ A \wedge B \end{array}$$

form a deduction of A,

$$\begin{array}{c} \vdots \\ \dfrac{A \wedge B}{A} \end{array}$$

and similarly with $B \wedge A$ in place of $A \wedge B$.

$\wedge$ – elimination.
From a deduction of $A \wedge B$

$$\begin{array}{c}\vdots\\ A \wedge B\end{array}$$

form a deduction of B,

$$\frac{\begin{array}{c}\vdots\\ A \quad \wedge \quad B\end{array}}{B}$$

and similarly with $B \wedge A$ in place of $A \wedge B$.

$\rightarrow$ -- introduction.
From a deduction of B with assumption A

$$\begin{array}{c}A\\ \vdots\\ B\end{array}$$

form a deduction of $A \rightarrow B$,

$$\frac{\begin{array}{c}\cancel{A}\\ \vdots\\ B\end{array}}{A \rightarrow B}$$

where none or some or all occurences of assumption A may be canceled.

$\rightarrow$ -- elimination.
From a deduction of A

$$\begin{array}{c}\vdots\\ A\end{array}$$

and a deduction of $A \rightarrow B$

$$\begin{array}{c}\vdots\\ A \rightarrow B\end{array}$$

form a deduction of B

$$\frac{\begin{array}{ccc}\vdots & & \vdots \\ A & A \to B\end{array}}{B}$$

Exercises. Give natural deduction proofs. (Here $\varphi \leftrightarrow \gamma$ is an abbreviation for $(\varphi \to \gamma) \wedge (\gamma \to \varphi)$.)

1. $(\varphi \wedge \varphi) \leftrightarrow \varphi$
2. $(\varphi \wedge \psi) \leftrightarrow (\psi \wedge \varphi)$
3. $((\varphi \wedge \psi) \wedge \sigma) \leftrightarrow (\varphi \wedge (\psi \wedge \sigma))$
4. $\varphi \to \varphi$
5. $\varphi \to (\psi \to \varphi)$
6. $(\varphi \to \psi) \to ((\psi \to \sigma) \to (\varphi \to \sigma))$
7. $((\varphi \to (\psi \to \sigma)) \to ((\phi \wedge \psi) \to \sigma)$
8. $((\phi \wedge \psi) \to \sigma) \to ((\varphi \to (\psi \to \sigma))$

∀ – introduction.
From a deduction of A

$$\begin{array}{c}\vdots \\ A\end{array}$$

form a deduction of $(\forall x)A$

$$\frac{\begin{array}{c}\vdots \\ A\end{array}}{(\forall x)A}$$

provided x is not free in any uncanceled assumption.

∀ – elimination.
From a deduction of $(\forall x)A$

$$\begin{array}{c}\vdots\\ (\forall x)A\end{array}$$

form a deduction of $A(t)$

$$\dfrac{\begin{array}{c}\vdots\\ (\forall x)A\end{array}}{A[t/x]}$$

provided t is a term free for x in A. (t is called free for x in A if the result $A[t/x]$ the simultaneous substitution of t for all free occurrences of x throughout A does not render any occurrence of any variable bound that was not bound before the substitution).

Example. We take an ordinary mathematical proof of $(\forall x)(\forall y)A(x,y) \rightarrow (\forall y)(\forall x)A(x,y)$ and convert it to a natural deduction proof.

ORDINARY PROOF	TRANSLATION
Suppose $(\forall x)(\forall y)A(x, y)$.	premise
Then $(\forall y)A(x, y)$.	$\forall$–elimination
So $A(x, y)$.	$\forall$–elimination
So $(\forall x)A(x, y)$	$\forall$–introduction
So $(\forall y)(\forall x)A(x, y)$.	$\forall$–introduction
Therefore $(\forall x)(\forall y)A(x, y) \rightarrow (\forall y)(\forall x)A(x, y)$.	$\rightarrow$–introduction and assumption cancellation

This gives the following natural deduction proof.

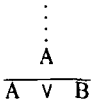
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cancel{(\forall x)(\forall y)A(x,y)}}{(\forall y)A(x,y)}}{A(x,y)}}{(\forall x)A(x,y)}}{(\forall y)(\forall x)A(x,y)}}{(\forall x)(\forall y)A(x,y) \rightarrow (\forall y)(\forall x)A(x,y)}$$

Exercise. Prove by natural deduction.

$[(\forall x)(\varphi(x) \wedge \psi(x))] \leftrightarrow [(\forall x)\varphi(x) \wedge (\forall x)\psi(x)]$

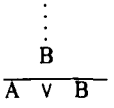
$\vee$ – introduction. (left)
From a deduction of A

form a deduction of $A \vee B$

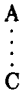


$\vee$ – introduction. (right)
From a deduction of B

form a deduction of $A \vee B$



$\vee$ – elimination. (right)
From a deduction of C from A



and a deduction of C from B

B
⋮
C

form a deduction of C

$$\dfrac{A \vee B \qquad \begin{array}{c}\overline{A}\\ \vdots\\ C\end{array} \qquad \begin{array}{c}\overline{B}\\ \vdots\\ C\end{array}}{C}$$

where none, some, or all occurrences of A, B as assumptions may be canceled, and $A \vee B$ is made an assumption.

The format of this rule is the hardest to get used to. But it is merely a notation for proof by exhaustion of cases. Read it as follows.

Case 1. Deduce C from assumption A.
Case 2. Deduce C from assumption B.
Therefore we may deduce C from assumption $A \vee B$, canceling premises A, B if we like.

Example. We give an example of translating ordinary proofs. Deduce conclusion $(A \vee B) \wedge (A \vee C)$ from premise $A \vee (B \wedge C)$.

ORDINARY MATHEMATICAL PROOF. Start from the conclusion and work back toward the premise.

Part 1. Suppose we know the conclusion $(A \vee B) \wedge (A \vee C)$. Then we must have had $(A \vee B)$ and $(A \vee C)$ as intermediate premises. This gives us two intermediate conclusions to work toward, $A \vee B$ and $A \vee C$.
Now start with the premise and work forward toward the conclusion.

Part 2. Suppose the premise $A \vee (B \wedge C)$ holds.
Case 1. Assume A. Then $A \vee B$.
Case 2. Assume $B \wedge C$. Then B, *and therefore* $A \vee B$.
Therefore from the premise $A \vee (B \wedge C)$ alone, we conclude $A \vee B$.

Part 3. Suppose the premise $A \vee (B \wedge C)$

Case 1. Assume A. Then $A \vee C$.

Case 2. Assume $B \wedge C$. Then C, and therefore $A \vee C$.

So we get $A \vee C$, based only on the premise $A \vee (B \wedge C)$. Therefore from the premise $A \vee (B \wedge C)$ alone we conclude $A \vee C$.

This completes the proof.

Part 1 is reflected by the $\wedge$–introduction.

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A \vee B & A \vee C\end{array}}{(A \vee B) \wedge (A \vee C)}$$

Part 2 is reflected by

$$\dfrac{A \vee (B \wedge C) \quad \begin{array}{cc}\cancel{B \wedge C} & \\ A & B \\ \overline{A \vee B} & \overline{A \vee B}\end{array}}{A \vee B}$$

Part 3 is reflected by

$$\dfrac{A \vee (B \wedge C) \quad \begin{array}{cc}\cancel{B \wedge C} & \\ A & C \\ \overline{A \vee C} & \overline{A \vee C}\end{array}}{A \vee C}$$

Combining,

$$\dfrac{\dfrac{A \vee (B \wedge C) \quad \begin{array}{cc}\cancel{B \wedge C} & \\ A & B \\ \overline{A \vee B} & \overline{A \vee B}\end{array}}{A \vee B} \qquad \dfrac{A \vee (B \wedge C) \quad \begin{array}{cc}\cancel{B \wedge C} & \\ A & C \\ \overline{A \vee C} & \overline{A \vee C}\end{array}}{A \vee C}}{(A \vee B) \wedge (A \vee C)}$$

First we decode the conclusion by introduction rules from intermediate premises to see

what we have to work forward to. Then we decode the original premises by elimination rules get the intermediate premises. Then the natural deduction proof consists of joining these pieces, one for the conclusion, one for every intermediate premise uncovered. This is the best method for finding natural deductions.

Example. Check yourself that the following is a derivation of $(A \wedge B) \vee (A \wedge C)$ from $A \wedge (B \vee C)$.

$$\dfrac{B \vee C \qquad \dfrac{\dfrac{\dfrac{A \wedge (B \vee C)}{A}\,\neg B}{A \wedge B}}{(A \wedge B) \vee (A \wedge C)} \qquad \dfrac{\dfrac{\dfrac{A \wedge (B \vee C)}{A}\,\neg C}{A \wedge C}}{(A \wedge B) \vee (A \wedge C)}}{(A \wedge B) \vee (A \wedge C)}$$

Example. We translate an ordinary proof into natural deduction. We deduce $(\forall x)((\varphi(x) \vee \psi(x))$ from $(\forall x)\varphi(x) \vee (\forall x)\psi(x)$.

ORDINARY PROOF.
There are two cases.
Case 1. $(\forall x)\varphi(x)$. In this case $\varphi(x)$ for any x, so $\varphi(x) \vee \psi(x)$.
Case 2.$(\forall x)\psi(x)$. In this case $\psi(x)$ for any x, so $\varphi(x) \vee \psi(x)$.
But we have assumption $(\forall x)\varphi(x) \vee (\forall x)\psi(x)$, so one of the cases holds. We can cancel the assumptions $(\forall x)\varphi(x)$, $(\forall x)\psi(x)$ and conclude $\varphi(x) \vee \psi(x)$, and therefore $(\forall x)(\psi(x) \vee \psi(x))$. The corresponding natural deduction is

$$\dfrac{\dfrac{(\forall x)\varphi(x) \vee (\forall x)\psi(x) \qquad \dfrac{\dfrac{\neg(\forall x)\varphi(x)}{\varphi(x)}}{\varphi(x) \vee \psi(x)} \qquad \dfrac{\dfrac{\neg(\forall x)\psi(x)}{\psi(x)}}{\varphi(x) \vee \psi(x)}}{\varphi(x) \vee \psi(x)}}{(\forall x)(\varphi(x) \vee \psi(x))}$$

Exercises. Give natural deduction proofs. Treat "if and only if", that is "$\leftrightarrow$", as defined by two implications.
1 $(\varphi \vee \psi) \leftrightarrow (\psi \vee \varphi)$
2. $((\varphi \vee \psi) \vee \sigma) \leftrightarrow (\varphi \vee (\psi \vee \sigma))$
3. $(\varphi \vee (\psi \wedge \sigma)) \leftrightarrow ((\varphi \vee \psi) \wedge (\varphi \vee \sigma))$
4. $(\varphi \wedge (\psi \vee \sigma)) \leftrightarrow ((\varphi \wedge \psi) \vee (\varphi \wedge \sigma))$

∃ – introduction.
From a deduction of A(t),

$$\vdots$$
$$A(t)$$

form a deduction of $(\exists x)A(x)$,

$$\vdots$$
$$\frac{A(t)}{(\exists x)A(x)}$$

provided t is free for x in A(x)

∃ – elimination.
From a deduction of B from A(x)

$$A(x)$$
$$\vdots$$
$$B$$

and provided x is not free in B or any assumption,
form a deduction of B with $(\exists x)A(x)$ as an assumption, and in which none or some or all occurrences of A(x) as assumptions may be canceled

$$\frac{(\exists x)A(x) \qquad \begin{array}{c}\cancel{A(x)}\\ \vdots \\ B\end{array}}{B}$$

Exercises. Give natural deduction proofs. Treat "if and only if" as defined by two implications.

1. $(\exists x)(\varphi(x) \vee \psi(x)) \leftrightarrow (\exists x)\varphi(x) \vee (\exists x)\psi(x)$
2. $(\varphi \vee (\forall x)\psi(x)) \rightarrow (\forall x)(\varphi \vee \psi(x))$, x not free in φ
3. $(\varphi \wedge (\exists x)\psi(x)) \rightarrow (\exists x)(\varphi \wedge \psi(x))$, x not free in φ
4. $(\exists x)(\varphi \rightarrow \psi(x)) \rightarrow (\varphi \rightarrow (\exists x)\psi(x))$, x not free in φ

5. $(\exists x)(\varphi \wedge \psi(x)) \dashv (\varphi \wedge (\exists x)\psi(x))$, x not free in φ
6. $(\exists x)(\varphi(x) \rightarrow \psi) \leftrightarrow ((\forall x)\varphi(x) \rightarrow \psi)$, x not free in ψ
7. $((\exists x)\varphi(x) \rightarrow \psi) \rightarrow (\forall x)(\varphi(x) \rightarrow \psi)$, x not free in ψ

Now assume that F (falsehood) is an additional logical constant.

$\neg$ – introduction (The absurdity rule)
From a deduction of F from A

$$\begin{array}{c} A \\ \vdots \\ F \end{array}$$

construct the deduction of B

$$\frac{\begin{array}{c} A \\ \vdots \\ F \end{array}}{B}$$

Remark. A particular case of this rule when $\neg A$ is substituted for B is that if A leads to an absurdity (that is, F), then $\neg A$ can be deduced. Think of $\neg A$ as meaning that the assumption A leads to a contradiction. In fact we can replace $\neg A$ by $A \rightarrow F$, and dispense with $\neg$ as a primitive entirely. This completes a system for intuitionistic propositional and predicate logic.

Exercise. Give intuitionistic natural deduction proofs. Treat if and only if as two implications.

1. $\neg\varphi \leftrightarrow \neg\neg\neg\varphi$
2. $(\varphi \wedge \neg\psi) \rightarrow \neg(\varphi \rightarrow \psi)$
3. $(\varphi \rightarrow \psi) \rightarrow (\neg\neg\varphi \rightarrow \neg\neg\psi)$
4. $(\neg\neg(\varphi \rightarrow \psi) \leftrightarrow (\neg\neg\varphi \rightarrow \neg\neg\psi)$
5. $\neg\neg(\varphi \wedge \psi) \leftrightarrow (\neg\neg\varphi \wedge \neg\neg\psi)$
6. $\neg\neg(\forall x)\varphi(x) \rightarrow (\forall x)\neg\neg\varphi(x)$
7. $\neg(\varphi \vee \psi) \leftrightarrow (\neg\varphi \wedge \neg\psi)$
8. $(\neg\varphi \wedge \neg\psi) \rightarrow \neg(\varphi \vee \psi)$

9. $(\neg\varphi \vee \neg\psi) \rightarrow \neg(\varphi \wedge \psi)$
10. $\varphi \rightarrow \neg\neg\varphi$
11. $F \leftrightarrow (\varphi \wedge \neg\varphi)$
12 $\neg(\exists x)\varphi(x) \rightarrow (\forall x)\neg\varphi(x)$
13. $(\forall x)\neg\varphi(x) \rightarrow \neg(\exists x)\varphi(x)$
14. $(\exists x)\neg\varphi(x) \rightarrow \neg(\forall x)\varphi(x)$

Classical natural deduction. To get a system for classical propositional or predicate logic, add the law of the excluded middle.

$\neg$ – elimination. (Reductio Ad Absurdum, or RAA)
From a deduction of falsity from $\neg A$

$$\begin{array}{c} \neg A \\ \vdots \\ F \end{array}$$

form a deduction of A

$$\dfrac{\begin{array}{c} \cancel{\neg A} \\ \vdots \\ F \end{array}}{A}$$

and none or some or all $\neg A$ assumptions can be canceled.

Remark. The absurdity rule would deduce from

$$\begin{array}{c} \neg A \\ \vdots \\ F \end{array}$$

only $\neg\neg A$. RAA deduces A.

Exercise. Show that every instance of the excluded middle $A \vee \neg A$ can be deduced using RAA.

Exercises. Below is a list of classically valid formulas which are not intuitionistically valid. For each give a natural deduction proof using the law of the excluded middle.

1. $(\varphi \vee \neg\varphi)$
2. $(\neg\neg\varphi \rightarrow \varphi)$
3.. $\neg(\varphi \wedge \psi) \rightarrow (\neg\varphi \vee \neg\psi)$.
4. $\neg\varphi \vee \neg\neg\varphi$
5. $(\varphi \rightarrow \psi) \vee (\psi \rightarrow \varphi)$
6. $(\neg\neg\varphi \rightarrow \varphi) \rightarrow (\varphi \vee \neg\varphi)$
7. $(\neg\varphi \rightarrow \neg\psi) \rightarrow (\psi \rightarrow \varphi)$
8. $(\varphi \rightarrow \psi) \rightarrow (\neg\varphi \vee \psi)$
9. $(\neg\varphi \vee \psi) \rightarrow (\varphi \rightarrow \psi)$
10. $\neg(\forall x)\varphi(x) \rightarrow (\exists x)\neg\varphi(x)$
11. $(\forall x)\neg\neg\varphi(x) \rightarrow \neg\neg(\forall x)\varphi(x)$
12. $(\forall x)(\varphi \vee \psi(x)) \rightarrow (\varphi \vee (\forall x)\psi(x))$, x not free in φ.
13. $((\varphi \rightarrow (\exists x)\psi(x)) \rightarrow (\exists x)(\varphi \rightarrow \psi(x))$, x not free in φ.
14. $((\forall x)\varphi(x) \rightarrow \psi) \rightarrow (\exists x)(\varphi(x) \rightarrow \psi)$, *x not free in* ψ.
15. $((\forall x)(\varphi(x) \vee \neg\varphi(x)) \wedge \neg\neg(\exists x)\varphi(x)) \rightarrow (\exists x)\varphi(x)$

§3. Heyting's semantics. In 1936 Heyting gave a sketch of an informal semantics for the logical connectives "$\wedge$","$\vee$","$\neg\rightarrow$","$\forall$" "$\exists$", "$\neg$", "F" based on the notion of a construction. His thought was that if one starts out with constructions of objects asserted to exist in the premises of an intuitionistic deduction, and an object is asserted to exist in the conclusion of the deduction, there should be a means of converting the deduction into a construction of the latter object. Heyting was not thinking in terms of an implementation of constructions as programs to be run on real hardware, none existed at that time. This was a conceptual exercise in construction, perhaps thought of in terms of pencil and paper.

Here are some of the issues. In classical mathematics, when a proposition is proven, it can be used as a lemma for all subsequent propositions without referring to the specific way the proposition has been proved. The proposition's past history, so to speak, plays no role in subsequent deductions based on the proposition. One proof of the proposition is as good as another, and need not be used later when using the proposition. In classical propositional logic this is reflected by the truth functionality (with T, F as the only truth values) of the logical connectives, that is the truth or falsity of the compound proposition is determined by truth or falsity of the parts and without reference to any further features of the parts. In Heyting's view the situation is different for intuitionistic logic. The intent is

to combine constructions establishing the parts of a compound proposition to get a construction establishing the compound proposition itself. This entails constructions which yield new constructions from old. This is like a construction for composition of functions which, applied to a construction for $f(x)=xx$ and a construction for the function $g(x)=x+2$ yields a construction for $g(f(x))=xx+2$. Further, different constructions associated with parts of a compound statement can be combined in different ways to give different constructions for the compound statement. We assume as did Heyting an informal notion of a construction as a rule which applies to constructions to yield constructions.

Here are some requirements that Heyting imposed on the notion of construction for interpreting intuitionistic logic.

1) A definition for atomic statements $P = R(c_1, ..., c_k)$ of the phrase "construction c proves P."

This c should be a construction showing that $R(c_1, ..., c_k)$ is true. For example, the atomic statement P might say that with a given definition of π, $\pi > 3$. A construction c such that c proves P should verify $\pi > 3$ from the definition of π.

2) Construction c proves $(A \wedge B)$ if and only if
c is a pair of constructions (d, e) such that d proves A and e proves B.

The intent here is that we can recover constructions d, e from construction c and conversely.

3) Construction c proves $(A \vee B)$ if and only if
c is a pair (d, e) such that d is a natural number and if d is 0, e proves A, if d is non–zero, e proves B.

The intent here is that we can recover a specified one of d, e from c and what it proves, and conversely.

4) c proves $(A \rightarrow B)$ if and only if whenever d proves A, then c(d) proves B.

The idea here is that c is a construction which, applied to any construction d proving A, produces a construction c(d) proving B.

5) Construction c proves $(\forall x)P(x)$ if and only if whenever d is a construction proving that e is an element of the domain D over which x ranges, then c(d) constructs P(e).

So c transforms proofs of instances of elementhood in D to proofs of instances of the universal statement. Thus c contains a proof of $(\forall x \epsilon D)P(x)$.

Remark. D is discrete if

$$\text{for all } \varphi,\ (\forall x)(x \epsilon D \rightarrow \varphi(x) \vee \neg\varphi(x)).$$

In case D is discrete, we may without loss identify each element of D with a canonical construction of that element. Then 5) would require merely that c is a function on D mapping members e of D to proofs c(e) of P(e). This simplification is not satisfactory when, as in the case of the real numbers, there is no obvious choice or even no possible canonical choice of construction for each element of D, but rather many constructions of each element of D. But even for the natural numbers, certain authors (Martin–Löf, DeBruijn) would require that the statement "2^{10} is a natural number" be proved by unravelling the definition of 2^{10}. This is certainly a concern in computer science as well.

6) Construction c proves $(\exists x)P(x)$ if and only if

c is a pair (d, e) such that d is a construction of an element f of D and e proves P(f).

These six requirements are criteria which any adequate definition of construction should obey. They do not resolve the question: What is a construction?

– We did not specify a base step saying what "c constructs atomic statement S" means.

– We did not specify what is meant by a construction for the elements of a domain, a significant problem for uncountable domains such as the real numbers.

– For construction c to prove that d is an element of a domain D means that elements of domain D have potential descriptions by means of constructions and that c proves a specific candidate description meets the requirement of being in D.

– A rule of inference should reflect a construction that converts the constructions associated with the premises of the rule to a construction associated with the conclusion of the rule.

– A deduction, leading from assumptions to a conclusion, possibly through many applications of various rules of deduction, should reflect a construction leading from constructions of its premises to constructions of its conclusion.

There is an interesting literature on the theory of constructions as an independent axiomatic theory, see Goodman [1970, 1973].

Heyting Arithmetic (HA). Intuitionistic first order arithmetic (HA) is obtained by taking the predicate logic Peano axioms for the natural numbers based on 0, successor, addition, multiplication, and the usual axioms for equality, as axioms in intuitionistic predicate logic. Here addition and multiplication are assumed to satisfy their recursion equations and the induction axiom is assumed for all formulas φ. An integer is represented as the numeral S(S(S(...0))) in HA. A (total) function f on the integers is called provably recursive in HA if there is a formula $\varphi(x, y)$ such that $(\forall x)(\exists y)(\varphi(x, y))$ is provable in HA, and for all integers m, n, if **m**, **n** are the respective numerals in HA , then $f(m) = n$ iff $\varphi(\mathbf{m}, \mathbf{n})$ is provable in HA. The set of provably recursive functions of HA contains all functions of integers that arise in ordinary mathematical or computational practice.

§4. **Kleene's realizability for HA.** *Kleene's 1945 notion of recursive realizability for HA* may be thought of as an interpretation within classical mathematics in which Heyting's "constructions" are codes for programs computing recursive functions on Turing machines. Kleene's own motivation was not Heyting's work, but rather the notion of "incomplete communication of information about a proposition" suggested by Kleene's reading of Hilbert–Bernays. Think of each construction as operating on input data (which are themselves descriptions of constructions) to produce output data (which are themselves descriptions of constructions). If we identify constructions with programs for computing partial recursive functions, and identify these programs with their codes (Gödel numbers), each construction can be represented conceptually by a partial recursive function φ operating on codes of input constructions to produce codes of output constructions. In Kleene's notation for partial recursive functions, $\{e\}y$ is the value (if any) of the partial recursive function defined by the e–th Turing machine evaluated at argument y. Kleene construed $\{e\}y$ as the code of a construction resulting from applying the construction with

code $\{e\}$ to the construction with code y.

Suppose we identify Heyting's constructions with indices of partial recursive functions. To fit Heyting's idea that from intuitionistic proofs, one should be able to extract a construction for objects proved to exist, there should certainly be a procedure for extracting from an intuitionistic proof of a statement

$$(\forall x)(\exists y)\varphi(x, y)$$

a recursive function $y(x)$ such that $(\forall x)\varphi(x, y(x))$. That is, there should be a procedure to extract a program to compute a function $y(x)$ from an intuitionistic proof of $(\forall x)(\exists y)\varphi(x, y)$. Kleene's 1945 realizability does this, and is the earliest example of extraction of programs (Turing machine programs) from intuitionistic proofs. Below is Kleene's definition of "e realizes φ" in HA for e an integer and φ a statement. Here "e realizes φ" will be a statement in HA itself. The definition is by induction on the length of statements. Kleene describes e informally as an incomplete communication of information about φ. Constable, the originator of NuPRL, coined the phrase "e is evidence for φ". In this spirit, what should the base clause for atomic statements say? Since these are pure numerical equations and HA proves the true ones and refutes the false ones, these atomic statements can be regarded as needing no evidence, or as being verified by any evidence. So we may start with

Atomic statements.
For atomic φ, e realizes φ is φ.

Remarks. For atomic φ, "e realizes φ" is defined to be the statement φ itself. This is a so–called "internal definition" of realizability by means of formulas within HA. The other clauses can be explained by the same words as were used for Heyting's constructions if "construction" is replaced by "index e of a partial recursive function $\{e\}$". Let p be a simple 1–1 recursive pairing function. Here p assigns a natural number $z = p(x, y)$ to each pair of natural numbers x, y. The components x and y will be denoted by $(z)_0 = x$, $(z)_1 = y$.

Conjunction.
e realizes $\varphi \wedge \psi$ is

$[(e)_0$ realizes $\varphi] \wedge [(e)_1$ realizes $\psi]$

Disjunction.
e realizes $(\varphi \vee \psi)$ is

$((e)_0 = 0 \rightarrow (e)_1$ realizes $\varphi) \wedge ((e)_0 \neq 0 \rightarrow (e)_1$ realizes $\psi)$

Implication.
e realizes $(\varphi \rightarrow \psi)$ is

$(\forall y)$(y realizes $\varphi \rightarrow \{e\}y$ is defined and realizes ψ)

Existential Quantification.
e realizes $(\exists y)\varphi(y)$ is

$(e)_1$ realizes $\varphi((e)_0)$

Remark. This means unpairing x gives a zeroth coordinate which is a witness to the existential quantifier and a first coordinate which is evidence for that fact.

Universal quantification.
e realizes $(\forall y)\varphi(y)$ is

$(\forall y)(\{e\}y$ is defined and realizes $\varphi(y))$

There is a formula in HA expressing that $\{e\}y$ is defined, namely $(\exists u)T(e, y, u)$, where T is Kleene's T–predicate (Kleene [1952]). So all these clauses can be written down in HA.

For the next section, replace Kleene's notation $\{x\}y$, universally used in recursion theory, by the notation (xy), a partial binary operation on numbers. This notation (xy) corresponds to that of Haskell Curry's theory of "applicative structures".

§5. Untyped application. Shönfinkel (1924) and Curry (1930) independently formalized the idea of a functional rule directly. This is combinatory logic. Instead of taking the notion

of membership in sets as fundamental for mathematics, Curry took as fundamental the notion of a functional rule which may apply to any mathematical object as argument to give a mathematical object as value. The new wrinkle was that he took the proper objects of mathematics to be the functional rules themselves, contrary to the point of view of set theory in which sets are the proper mathematical objects. But since a functional rule is to operate on any mathematical object as argument, and mathematical objects are themselves the functional rules, it follows that each functional rule F can be applied to any functional rule G as argument, producing a functional rule (FG) as value. We have then a binary operation on rules, the operation of application.

We do not view the notion of application as a foundation for mathematics. Instead, we give the classical mathematical definition of an application structure within set theory, just like the definition of any other contemporary mathematical structure. This is not connected with Curry's original purpose, but is the way that many computer science applications arise. The notion of a function as a rule for assigning values to arguments was prevalent for centuries before the current set–theoretic notion of a function as single–valued set of ordered pairs was adopted as a definition by the mainstream of mathematics. After the turn of the twentieth century it became fashionable to define every mathematical object as a set, so then the word "function" was usually restricted to the notion of single valued set of ordered pairs.

Applicative structures. An (untyped) applicative structure will be a non–empty set C with a binary partial operation , called application, on elements F, G of C. This is written (FG) or even FG. Axioms are below. Intuitively, each element of an applicative structure C represents a code for a "functional rule". Curry invented untyped abstract application structures as defined above in the case that application is total. Feferman extended this to partial application. The best reading of the partial operation (FG) is as follows.

"F is the code of a functional rule which, when applied to an argument which is a code G of a functional rule, assigns as value the code (FG) of another functional rule".

Example (Kleene Structure). Let C be the set of non–negative integers with the Kleene operation $\{x\}(y)$, written as application (xy). Here there is no doubt what is intended by saying that x, y, (xy) represent functional rules. They are all code numbers for Turing programs for computing partial recursive functions of one variable. This operation (xy) is

partial since $\{x\}$ may not be defined at y.

Applicative terms. Let C be an applicative structure. The class of terms over the language of any C is defined inductively as follows.

(i) Every element of C is a term over C, used as its own constant name.

(ii) Variables X, Y, Z, ... are terms over C.

(iii) If α, β are terms over C, then $(\alpha\beta)$ is a term over C.

These are the natural "polynomials" in applicative structures. When writing terms and omitting parentheses, we assume left association.

Let $\tau(Y_1, ..., Y_n)$ be a term over C with at most the indicated free variables. Then τ induces a (partial) function on a subset of the n–tuples $(c_1, ..., c_n)$ *from* C *with values* in C, defined by substituting c_1 for Y_1, ..., c_n for Y_n in τ, and evaluating when defined. The left multiplications of C are the partial maps of C to C given by $x \to (cx)$ for a c in C. The applicative structure C is called functionally complete if for every term $\tau(x)$ with one free variable x, the partial function on C to C induced by τ is a left multiplication induced by an element c of C. This is the property we want to have for applicative structures. It is guaranteed by putting into C special elements.

Definition. An applicative structure (often called a partial applicative structure) consists of a set C and a partial operation (xy) on C such that C has distinct elements K and S and I for which

0) $Ia=a$

1) $Sxy = (Sx)y$ is always defined,

2) $Kxy = x$,

3) $Sxyz = ((Sx)y)z$ is defined iff $xz(yz)$ is defined, and then $Sxyz = xz(yz)$.

Note that I is dispensible in favour of SKK since for all x, $Ix = SKKx$, but we keep I anyway.

Exercise. Interpret I, K, S in Kleene's applicative structure with domain the natural

numbers and application operation $\{x\}x = xy$, and verify the axioms.

Example. In the Kleene structure the partial function $\tau(c_1, ..., c_n)$ is partial recursive and an index for this partial recursive function can be computed from τ. Thus each term with one free variable $\tau(Y)$ induces a partial recursive function on C to C with index x. This means that the function induced on C is left multiplication by x.
So the Kleene structure is functionally complete.

We want to think of elements of C as functional programs whose inputs and outputs are functional programs. The following construction is one way of making this possible. In any applicative structure we define λ–abstraction.

Theorem. For each term t over C built up from variables and application and constants K, S, I, there is a term over C, which we abbreviate as $\lambda X.t$, whose free variables are those of t leaving out X such that

$\lambda X.t$ is always defined, and

$(\lambda X.t)X$ is defined iff t is defined and these values are equal.

Proof. Curry's inductive definition of λ is as follows.

$(\lambda X).X$ is SKK, or I.
$(\lambda X).t$ is Kt if t is a constant or a variable other than X,
$(\lambda X).uv$ is $S((\lambda X).u)((\lambda X).v)$.

Exercise. Verify the theorem above and conclude functional completeness of all applicative structures C.

§6. Untyped λ–calculus. In the early 1930's Alonzo Church developed another theory of functional rules covering much the same ground, expressed as "untyped λ–terms". We do not give a modern discussion of general notions of untyped λ–calculus analogous to the general notion of a partial combinatory structure discussed above, but restrict ourselves for lack of space to Church's original case. We outline only the definition of λ–term, and how to use such terms as functional rules to compute functions of integers.

The syntax consists of variables "X_0, X_1, ...," the abstraction symbol "λ", and parentheses "(", ")", and dot ".".

Untyped λ–terms.

i) A variable X is a λ–term. The sole occurrence of X in X is free.

ii) If M, N are terms, then the application (MN) of M to N is a term. Occurrences of variables are free or bound in (MN) as they are in M, N.

(iii) If M is a term and X is a variable, then the λ–abstraction $(\lambda X).M$ of M with respect to X is a term. An occurrence of a variable other than X in $(\lambda X).M$ is free or bound as it is in M. Occurrences of X in $(\lambda X).M$ are all bound.

Informally the standard interpretation is that

– variables range over λ–terms

– λ–abstraction. $(\lambda X).t$ is a code for the rule making the rule with code t a function of X.

– application. (MN) is the application of rule with code M to argument with code N to give as value the code (MN) of another rule.

Of course, the notation also admits other interpretations.

The convention is that $\lambda X_1...X_n.M$ is an abbreviation for $\lambda X_1(...(\lambda X_n.M)...)$, making the rule with code M a function of X_1, ..., X_n. Also $M_1M_2...M_n$ is the abbreviation for $(...(M_1M_2)...M_n)$.

Reduction Rules.

α–rule. We can rename bound variables, that is

$\lambda x.M$ immediately reduces to $\lambda Y.M[X/Y]$, where $M[X/Y]$ is the result of substituting X

for Y everywhere in M.

β–rule. $(\lambda X.M)N$ immediately reduces to $M[X/N]$,
provided every free occurrence of variables in N before substituion remains free after the *substitution.*

This can always be achieved by applying a suitable α–reduction before the intended β–reduction.

Reducibility.
A λ–term s is reducible to a λ–term t if there is a sequence of λ–terms $s = t_0, t_1, ..., t_n = t$ such that each t_{i+1} is obtained from t_i by replacing a subterm t_i' of t_i by a term immediately reducible to t_i using an α–rule or a β–rule. A term is normal if there is no subterm to which the β–rule can be applied. A term is normalizable if it is reducible to a normal term, called a normal form for that term.

Examples. Normal forms of terms do not necessarily exist. Let Δ be $\lambda x.xx$, Δt β–reduces to tt, $\Delta\Delta$ β–reduces to $\Delta\Delta$. Even for terms with normal forms, there can be infinite sequences of looping β–reductions. Look at $(\lambda XY.Y)(\Delta\Delta)\alpha$, which β–reduces to α, and also to $(\lambda XY.Y)(\Delta\Delta)\alpha$, depending on whether the whole term or the subterm Δ is immediately β–reduced.

Theorem. (Church–Rosser) If a term has a normal form, the form is unique.

Church also proved there is no recursive procedure for telling whether or not a term has a normal form.

Here is the *usual representation of an integer* n as a term $\underline{n}$ in the Church λ–calculus. Identify integer n with a term $\underline{n}$ representing n–fold iteration of a function f, that is, $\underline{n}fX = f^{(n)}(X)$, where $f^{(n)}(X)$ is $f(f(...f(X)...))$, f iterated n times. So $\underline{n}$ should be $\lambda YX.Y^{(n)}X$, the "Church numeral" for n. So zero is represented by the λ–term $\lambda YX.X$, and successor is represented by $\lambda UYX.Y(UYX)$. The Church numerals are normal terms, hence by the Church–Rosser theorem they cannot be reduced to one another. The execution of a computation in the Church λ– calculus is a sequence of immediate reductions applied to a λ –term. The computation terminates when a normal form is

obtained. The normal form is regarded as a code for the intended result of the computation. If the aim is to compute an integer with code a Church numeral by reducing a term to normal form, the normality of Church numerals guarantees a unique integer result if any.

The usual method of computing partial functions f of integers in Church's λ–calculus is based on this. A partial function f (of n integer variables) is called λ–definable if there is a λ–term F such that for all Church numerals $a_1, ..., a_n$, b,

$$f(a_1, ..., a_n) = b \text{ iff } F\underline{a}_1 ... \underline{a}_n \text{ is reducible to } \underline{b}.$$

That is, to compute the value $f(a_1, ..., a_n)$, we reduce the term $F\underline{a}_1 ... \underline{a}_n$ to normal form $\underline{b}$, a Church numeral. Then $f(a_1, ..., a_n)$ is the integer b.

Theorem. (Kleene) Every partial recursive function is λ–definable, and conversely.

We give only a simple example of a function definition. Definition by cases can be defined by first defining truth values $T = \lambda X.\lambda Y.X$ and $F = \lambda X.\lambda Y.Y$ (analogous to T = 0 and F = 1 in ordinary logic). Then, if f, g are λ–terms and b takes values T, F, and we set Dfgb = (bf)g, it is easy to check that DfgT reduces to f and DfgF reduces to g. Notice however that Dfgb represents a (possibly partial) computable function even when b reduces to neither T nor F.

To see that this original Church λ–calculus gives a Curry combinatorial structure, set

$I = \lambda x.x$
$S = \lambda xyz.xz(yz)$
$K = \lambda xy.x.$

Note. Ix is SKKx.

§7. Set theory and application. Curry intended combinatory structures as a formal basis of the necessary properties of "functional rule". Because of the way mainstream mathematics is written, it may appear to the naive that the notion of a function as single valued set of ordered pairs has completely supplanted the notion of a functional rule. In set theory an example of a functional rule which is not a function is the rule $\lambda x.y$

assigning to each set x its unit set $y = \{x\}$. If this were a function, its domain would be a set, while it is the proper class of all sets. We review the foundations of set theory as established by Zermelo, Fraenkel, and Skolem in the period 1908–1920. The primitive notions of Zermelo–Fraenkel set theory are set and membership. So the primitive relation is ϵ, the quantifiers range over all sets. Equality of sets $x = y$ can be defined as $(\forall z)(z \epsilon x \leftrightarrow z \epsilon y)$. (What makes ZF extensional is the assertion that this notion of equality is "true" equality in the sense that it satisfies the equality axioms $x = y \rightarrow \varphi(x) = \varphi(y)$ for all φ.) Set theory axioms assert that certain initial sets exist and provide rules for the construction of new sets from old. For example,

$(\exists z)(\forall x)\neg(x \epsilon z)$

asserts there is an empty set z, which we abbreviate as ϕ.

$(\forall x)(\forall y)(\exists z)(\forall w)(w \epsilon z \leftrightarrow w = x \vee w = y)$

asserts there is an unordered pair z with members x, y, which we abbreviate as $\{x, y\}$ (so $\{x\}$ is $\{x, x\}$).

Note: the usual definition of ordered pair (x, y) entering into the definition of function is ordered pair $(x, y) = \{\{x\}, \{x, y\}\}$.

$(\forall x)(\exists y)((\forall w)(w \epsilon y \leftrightarrow (\exists z)(w \epsilon z \wedge z \epsilon x))$

asserts that the set of sets x has a union y, which we abbreviate as $\cup x$. Then we define ordinary union $x \cup y$ as $\cup \{x, y\}$.

$(\forall x)(\exists y)(\forall z)(z \epsilon y \leftrightarrow (\forall u)(u \epsilon z \rightarrow u \epsilon x))$

asserts that set x has power set y consisting of all subsets of x.

Let $P(x)$ assert that the null set is a member of x, and for all y, whenever $y \epsilon x$, then $y \cup \{y\} \epsilon x$. The axiom of infinity says there is a least x satisfying $P(x)$, and this x is denoted by ω, the set of integers, and $x+1 = x \cup \{x\}$.

Axiom of Foundation. Any non–empty set has an element disjoint from it.

The crucial axiom is replacement, where functional rules rear their heads in set theory.

Axiom of Replacement. Let A be a set and let φ be a formula $\varphi(x_1, ..., x_n, x, y)$ in the language of set theory such that for given values of $x_1, ..., x_n$,

$$\varphi(x_1, ..., x_n, x, y) \wedge \varphi(x_1, ..., x_n, x, z) \rightarrow y = z.$$

Then there is a set B such that $y \in B$ if and only if there exists an x in A such that $\varphi(x_1, ..., x_n, x, y)$. That is, the image of a set under the functional rule given by φ is a set.

Suppose the formula φ is $x = y \wedge \psi(x, x_1, ..., x_n)$. Then we get as a special case of replacement the

Axiom of Separation. Given any set A, any formula $\psi(x, x_1, ..., x_n)$, any values of parameters $x_1, ..., x_n$, there is a set B such that for all x, $x \in B$ iff $\psi(x, x_1, ..., x_n)$.

(In "pure Zermelo" set theory we keep the axiom of separation and drop replacement.)

The axiom of replacement is necessary for the development of mathematics, in particular to justify definitions by transfinite induction. But such a formula φ in replacement generally does not define a single valued set of ordered pairs. Consider as above the formula $\varphi(x, y)$ that says $y = \{x\}$. If there were a single valued set of ordered pairs F such that $\varphi(x, y)$ iff $y = \{x\}$, the domain of F would be a a set V. This would be the set V of all sets, and would lead directly to Russell's contradictory set $\{x \in V: \neg x \in x\}$ and to a contradiction. So V is not a set and $\varphi(x, y)$ does not define a function as a single valued set of ordered pairs in Zermelo–Fraenkel set theory. Thus functional rules enter into the foundations of set theory through the axiom of replacement. So formulas with set parameters act like Curry's functional rules.

This is more than an analogy. Each formula with set parameters can be coded as a set by taking as its code a finite sequence consisting of the Gödel number of the formula and its set parameters, and conversely every set can be thought of as a code of such a formula with set parameters. So if F, G are codes of formulas with set parameters, a partial

application operation on formulas with set parameters F, G can be defined by letting (FG) be the set assigned by the functional rule with code F evaluated at G.

§8. **Simple typed λ–calculus.** Typed λ–calculi are a return to the modern (non–self–applicative) notion of function, where domain and range are quite explicit as types. In computer science types arose first to enforce programming discipline and avoid errors by typing. They turn out to have a much more profound interest (see the next section).

Simple Types.
Types are generated by the following inductive clauses.

(i) Type constants α, β, γ, ... are types (atomic types).

(ii) If σ and τ are types, then $(\sigma \Rightarrow \tau)$ is a type (exponential types).

(iii) If σ, τ are types, then $(\sigma \times \tau)$ is a type (product types).

Terms.
We begin with a list of variables of type σ x_1^σ, x_2^σ, x_3^σ,....
Then the terms of type σ are generated by the following inductive clauses.

(i) Variables of type σ are terms of type σ.

(ii) If s, t are terms of respective types σ, τ, then $(s \times t)$ is a term of type $(\sigma \times \tau)$. An occurrence of a variable is free in $s \times t$ if and only if free in whichever of s, t it occurs.

(iii) If t is a term of type $(\sigma \times \tau)$, then $\pi^1 t$ is a term of type σ, $\pi^2 t$ is a term of type τ.

An occurrence of a variable is free in $\pi^1 t$ or $\pi^2 t$ if and only if free in t.

(iv) If t, u are terms of respective types $(\sigma \Rightarrow \tau)$, σ, then (tu) is a term of type τ.

An occurrence of a variable in (tu) is free if and only if free in whichever of t, u it occurs.

(v) If t is a term of type τ and if x is a variable of type σ, then $\lambda x.t$ is a term of type $(\sigma \Rightarrow \tau)$.

An occurrence in $\lambda x.t$ of a variable different from x is free if and only if free in t; all occurrences of x in $\lambda x.t$ are not free.

An occurrence of a variable x in a term t is bound if not free.

Reduction of λ–calculus terms to normal form was viewed from the beginning by Church as a computation procedure beginning with the term to be reduced, ending with a normal form for that term as the "answer" to a problem of computation. We state standard reduction results for the typed lambda calculus with types based on $\times$, $\Rightarrow$.

Reduction.

Let x be a variable of type σ, let u be a term of type σ. In this and only in this case define t[u/x] as the term resulting by substituting u for all free occurrences of x in t, where bound variables in t are systematically renamed so as to be distinct from the free variables in u. (It is easily shown by induction on the definition of the term t that t[u/x] is a term of the same type as t.)

Exercise. Define t[u/x] by induction on the length of the term t. In substituting new variables for bound variables, always use the first suitable variable of the same type.

Definition. The following three clauses define "term r contracts to term c".

(i) term $r = \pi^1(t \times u)$ contracts to term t.

(ii) term $r = \pi^2(t \times u)$ contracts to term u

(iii) term $r = (\lambda x^{\sigma}_{n} t)u$ contracts to term t[u/x], provided u is of type σ.

The left term r is called the redex, the right term c is called the contractum.

Definition. Term v is immediately reducible to term w if w results from v by replacing one occurrence of a subterm of t as redex by the contractum of that redex using one application of one of the three rules above.

A finite or infinite sequence of terms $t_0,\ldots, t_n, \ldots$ is a reducibility sequence if, for all i for which t_i is defined, t_{i-1} is immediately reducible to t_i.

Term v is reducible to term w if there is a finite reducibility sequence beginning with v, ending with w. We write this as $v > w$.

Definition. A term is is called a normal term if it contains no redex, that is no subterm of the form $\pi^1(t \times u)$, $\pi^2(t \times u)$, $(\lambda x^{\sigma}_n.t)u$.
(These are the terms for which no reduction is possible.)

Definition. A normalization of t is a finite reducibility sequence, beginning with t, ending with a normal term w. A term t is normalizable if there is a normalization starting with t.

Weak normalization theorem. (See Girard et al.[1989].) Every term is normalizable.

Definition. A term t is strongly normalizable if there is no infinite reducibility sequence starting with t.

A harder theorem is the

Strong normalization theorem. (See Girard et al. [1989].) All terms of the simple typed lambda calculus are strongly normalizable.

In the untyped λ–calculus, reduction of λ–terms to normal form was, as we have said earlier, Church's method of performing computations. This is a good view for the typed calculi as well, but in the simple typed λ–calculus every potentially infinite sequence of reduction steps terminates in a normal form due to strong normalization, and the collection of functions so computable is much smaller.

§9. Curry–Howard isomorphism. Another semantics of intuitionistic logic, which offers an

interpretation different from realizability of Heyting's theory of constructions, uses the typed lambda calculus and stems from Curry–Feys [1958]. There is is an isomorphism, called the Curry–Howard isomorphism, between two structures which were apparently unrelated before Curry. One is the calculus of deductions in the intuitionistic propositional logic. The other is a calculus of typed terms in a typed lambda calculus with types the propositions of that propositional logic. We discuss here the Curry–Howard isomorphism for the fragment of intuitionistic propositional logic based on $\wedge$, $\rightarrow$. The isomorphism maps

– propositions to corresponding types

– deduction of propositions to terms of the corresponding type.

Here

– atomic propositions correspond to type constants.

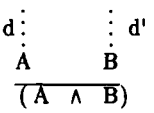
– compound proposition $A \wedge B$ corresponds to type $A \times B$

– compound proposition $A \rightarrow B$ corresponds to type $A \Rightarrow B$.

This $\wedge$, $\rightarrow$ fragment is also the basis of the cartesian closed category approach to intuitionistic logic. We follow Girard's notations for the most part. We now construct the Curry–Howard isomorphism formally by showing how to construct a typed λ–term from a deduction tree.

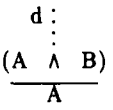
0) Assumptions.
Term x_n^A corresponds to an occurrence of assumption A at a leaf. There is a different subscript n for each such instance.

1) $\wedge$ – introduction.
If term t of type A corresponds to deduction d of formula A, and term t' of type B corresponds to deduction d' of formula B, then term $t \times t'$ is of type $A \times B$ and corresponds to the following deduction of $(A \wedge B)$



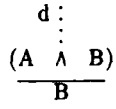
2a) ∧ – elimination. (left)
If term t is of type $A \times B$ and corresponds to deduction d of $(A \wedge B)$, then

term $\pi_1 t$ of type A corresponds to the following deduction of A

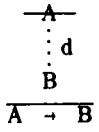


2b) ∧ – elimination. (right)
If t is a term of type $A \times B$ corresponding to a deduction d of $(A \wedge B)$, then

term $\pi_2 t$ of type B corresponds to the following deduction of B.



3a) ⇒ – introduction.
If t is a term of type B corresponding to a deduction d of B from assumption A, then $\lambda x^A.t$ is the term of type $(A \Rightarrow B)$ associated with the deduction



where some or all or no instances of A as assumptions are canceled.

3b) ⇒ – elimination.
If t of type A, t' of type $(A \Rightarrow B)$ are the terms corresponding to deductions d of A, d' of $(A \Rightarrow B)$, then (t't) is the term of type B corresponding to the following deduction of

B

$$\dfrac{\begin{array}{cc} d: & \\ \vdots & \\ A & \end{array} \qquad \begin{array}{cc} & d' \\ & \vdots \\ (A \to & B) \end{array}}{B}$$

The normalization theorem for simple typed λ–calculus corresponds under the Curry–Howard isomorphism to identical results on normalization of natural deductions in the propositional logic based on $\wedge$, $\to$. Under the Curry–Howard isomorphism, a reduction step for a term in the typed lambda calculus corresponds to elimination of a superfluous step in a corresponding natural deduction. Elimination of superfluous steps in deductions was a reworking of Gentzen's cut–elimination ([1935, 1969]) for use in consistency proofs for classical first order arithmetic. (But the $\wedge$, $\to$ fragment we have discussed corresponds to a tiny fragment of Gentzen's work.) He reduced all proofs of first order arithmetic step by step to a "cut–free" form by an algorithm for the elimination of "superfluous steps". Similarly, proofs can be reduced to a "normal" or "redundancy–free" form. One could see by finitistic reasoning that a "normal form" proof could not end in a contradiction. The whole force of the argument for consistency is to prove that simplification of a deduction by elimination of superfluous steps always terminates in a "normal form" proof. This is where, for full first order intuitionistic or classical arithmetic, non–finitary reasoning is used to show that a specific tree of height ϵ_0 is well–founded. This tree describes possible reduction sequences to normal form.

Normalization of deductions is implicit in the theorems of Curry and Feys [1958], and was set forth explicitly in Prawitz [1965]. The connections between cut–elimination and normalization, translations of one to the other, were made precise by Zucker [1974] and Pottinger [1977].

§10. Typed combinators. In the simple typed λ–calculus discussed above there are typed versions of the Curry combinators (see Hindley and Seldin [1986], Lambek [1980]),

$I_\alpha = (\lambda x^\alpha . x^\alpha)$,
$K_{\alpha,\beta} = (\lambda x^\alpha y^\beta . x^\alpha)$,
$S_{\alpha,\beta,\gamma} = (\lambda x^{\alpha\to\beta\to\gamma} y^{\alpha\to\beta} z^\alpha . xz(yz))$

Curry and Feys [1958] noted that these three typed combinators are formally analogous to the axiom schema of implicational propositional calculus

$A \to A$,
$(C \to (B \to A)) \to ((C \to B) \to (C \to A))$,
$A \to (B \to A)$.

They noted that a sequence of (legally applied) successive applications of three three typed combinators I, K, S correspond to the sequence of steps of a deduction using modus ponens as the rule of inference. This was the origin of the Curry–Howard isomorphism. We can think of typed combinators as "generalized deduction rules" for intuitionistic logics based on modus ponens and implication, and possibly other logical operations.

For simplicity of exposition, in this one section types will be based on "$\to$" alone (arrow types) , as in Curry–Feys [1958] and Hindley–Seldin [1986].

Arrow types.
1. Constant types (atomic types) are types.
2. If α and β are types, so is $(\alpha \to \beta)$.

We left–associate, $((\alpha \to \beta) \to \gamma)$ is $\alpha \to \beta \to \gamma$.

Typed combinatory terms.
For each type α, there are infinitely many variables " v^{α} ", constants "$K_{\alpha,\beta}$", "$S_{\alpha,\beta,\gamma}$", parentheses "(", ")".

1. Each v^{α}, $K_{\alpha,\beta}$, $S_{\alpha,\beta,\gamma}$ is a typed combinatory term of respective type α, $\alpha \to \beta \to \alpha$, $(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$.

2. If X and Y are typed combinatory terms of types $\alpha \to \beta$ and α respectively, then $(X^{\alpha \to \beta} Y^{\alpha})$ is a typed combinatory term of type β.

There is a notion of reduction, and a strong normalization theorem holds for typed combinatory terms. See Hindley and Seldin [1986]. But we have said enough to indicate what typed combinatory structures look like, and how they relate to typed λ–calculus.

§11. Second order intuitionistic logic. We now introduce a second order propositional calulus based on Λ, $\rightarrow$. There is an infinite list P_0, P_1, ... of "propositional variables".

i) Propositional variables are propositions.
ii) If α, β are propositions, then $(\alpha \rightarrow \beta)$ is a proposition.
ii) If α is a proposition, β is a propositional variable, then $\Lambda\beta.\alpha$ is a proposition.

This list of primitives suffices for second order logic, since all the other standard connectives are then definable. This is a very powerful theory, much stronger than it appears at first. This is the theory which gives rise, under a strengthening of the Curry–Howard isomorphism described above, to "polymorphic lambda calculus", or the system F of Girard, described briefly later.

Substitution. A substitution B[A/C] of proposition A for all occurences of variable C in proposition B is called legal if no variable occurrence free in A becomes bound in B[A/C]. The rules defining deductions are as follows.

Atomic deductions.
For every formula A

A

is a deduction. A is not canceled.

$\rightarrow$ – introduction.
From a deduction of B with premise A

A
⋮
B

form a deduction of $A \rightarrow B$,

$$\dfrac{\begin{array}{c}\cancel{A}\\ \vdots\\ B\end{array}}{A \to B}$$

where none or some or all occurrences of premise A may be canceled.

<u>→ – elimination</u>
From a deduction of A

$$\begin{array}{c}\vdots\\ A\end{array}$$

and deduction of $A \to B$

$$\begin{array}{c}\vdots\\ A \to B\end{array}$$

form a deduction of B,

$$\dfrac{\begin{array}{ccc}\vdots & & \vdots\\ A & A & \to\ B\end{array}}{B}$$

<u>Λ–introduction</u>.
From a deduction of B in which the propositional variable A does not occur free above the line

$$\dfrac{\begin{array}{c}\vdots\\ \end{array}}{B}$$

form a deduction of ΛA.B,

$$\frac{\begin{array}{c}\vdots\\ B\end{array}}{\Lambda A\,.\,B}$$

Λ – elimination.
From a deduction of $\Lambda A.B$,

$$\frac{\begin{array}{c}\vdots\\ \Lambda A\,.\,B\end{array}}{}$$

and a legal substitution $B[C/A]$, form a deduction of $B[C/A]$,

$$\frac{\begin{array}{c}\vdots\\ \Lambda A\,.\,B\end{array}}{B[C/A]}$$

§12. Polymorphic λ–calculus. Polymorphic lambda calculus is the typed calculus corresponding to the second order propositional calculus introduced above, with its *propositions* corresponding to types, its deductions corresponding to terms.

Girard (1972) introduced free type variables as atomic types, and also allowed types which have universally quantified type variables. In our notation for polymorphic terms, two kinds of variable occur. Term variables occur on the line and type variables occur in superscripts of terms, and on the line as well. It is convenient to define a term as a finite sequence of symbols, including the type variables. So we follow the convention that the non–linear notation x^{σ} is an abbreviation for the string $(-x\sigma)$. We put the "–" in to avoid syntactical confusion.

Remark. The intuition is something like this. Think of each x^{σ} as a box with label σ in which a term assigned to x may be stored, so long as its type is σ. Think of σ as a label with spaces (type variables) which can be filled out with types, thus changing the type of the x's that can be stored there. Since such type variables will then occur in the type expressions which are superscripts of variables, terms are functions of type variables.

There will be two additional primitive term–builders for the polymorphic calculus, beyond those for the simple typed calculus. One new term–builder of polymorphic calculus is type application, namely application of a term t to a type α to get a term $t\alpha$, analogous in

notation to the application of a term t to a term u to get a term tu in simple typed lambda calculus. The second new term builder of polymorphic calculus is type abstraction. If t is a term of type σ and α is a type variable, then $\lambda\alpha.t$ is a term of type $\Lambda\alpha.\sigma$, meaning that t has been made a function of the type variable α. This pair of term builders, type abstraction and type application, are the new term builders of polymorphic calculus added to simple typed lambda calculus, which has itself only term application and term abstraction as term builders.

Polymorphic types.

(i) Type variables α, β, γ, ... are types. The occurrence of a type variable in itself is free.

(ii) If σ and τ are types, then $(\sigma \to \tau)$ is a type. An occurrence of a type variable in $\sigma \to \tau$ is free or bound as it is in σ, τ respectively.

(iii) If σ is a type and α is a type variable, then $\Lambda\alpha.\sigma$ is a type. All occurrences of α in $\Lambda\alpha.\sigma$ are bound. An occurrence of a type variable other than α in $\Lambda\alpha.\sigma$ is free or bound as it is in σ.

We do not distinguish types that differ only in their bound variables. $\sigma[\tau/\alpha]$ is the result of substituting type τ for free occurrences of type variable α in type σ, where the bound variables of σ must be renamed so as to be distinct from the free type variables of τ.

Polymorphic Terms.

(i) For any type σ, the term variables of type σ,

x^σ, y^σ, z^σ, ...

are terms of type σ. In x^σ the occurrence of the term variable x^σ is free. An occurrence of a type variable in x^σ is free or bound as it is in σ.

(ii) If t is a term of type τ and x^σ is a term variable of type σ, then $\lambda x^\sigma.t$ is a term of type $\sigma \to \tau$. All occurrences of x^σ in $\lambda x^\sigma.t$ are bound. Occurrences of term variables other than x^σ in $\lambda x^\sigma.t$ are free or bound as they are in σ or in t.

(iii) If t, u are terms of respective types $(\sigma \to \tau)$, σ, then (tu) is a term of type τ.

Occurrences in tu are free or bound as they are in t, u. This (tu) is usually written t(u), read t applied to u (term application).

(iv) If t is a term of type σ and α is a type variable which does not occur free in the type of any term variable occurring free in t, then $\Lambda\alpha.t$ is a term of type $\Lambda\alpha.\sigma$. All occurrences of the type variable α in $\Lambda\alpha.t$ are bound. Occurrences of term variables or type variables other than α in $\Lambda\alpha.t$ are free or bound as they are in t.

(v) If t is a term of type $\Lambda\alpha.\sigma$ and if τ is a type, then $(t\tau)$ is a term of type $\sigma[\tau/\alpha]$. Occurrences of term or type variables in $(t\tau)$ are free or bound as they are in t, τ. This $(t\tau)$ is usually written $t(\tau)$, read term t applied to type τ (type application).

Note that (iv) and (v) are abstraction and application for type variables, just as (ii) and (iii) are abstraction and application for term variables. The reason for the limitation on α in (iv) will be clear from the (extended) Curry–Howard isomorphism below.

Reduction.
We use $t[u/x^{\sigma}]$ for the result of substituting u for all free occurrences of term variable x^{σ} in term t, where u is of type σ and the bound term variables in t are renamed so as to be distinct from the free term variables in u, t. Similarly $t[\tau/\alpha]$ is the result of substituting type τ for all free occurrences of type variable α in term t and the bound type variables are renamed so as to be distinct from the free type variables in τ, t.

We now define reducibility as in the simple typed calculus, but with different clauses.

Definition. The following two clauses define "term r contracts to term c".
Suppose u is any term of the same type as term variable x. Then
(i) term $r = (\lambda x.t)u$ contracts to term $t[u/x]$.
Suppose τ is any type of the same type as type variable α. Then
(ii) term $r = (\lambda\alpha.t)\tau$ contracts to term $t[\tau/\alpha]$.

The left term r is called the redex, the right term c is called the contractum. Clearly redex and contractum always have the same type.

Definition. Term v is immediately reducible to term w if w results from v by replacing one occurrence of a subterm of t as redex by the contractum of that redex using

one application of one of the two rules above.

A finite or infinite sequence of terms $t_0, ..., t_n, ...$ is a reducibility sequence if for all i for which t_{i-1} is defined, t_{i-1} is immediately reducible to t_i.

Term t is reducible to term w if there is a finite reducibility sequence beginning with v, ending with w. We write $v > w$. A term is normal if it has no redex. A normalization sequence starting with t is a finite reducibility sequence ending in a normal term. A term t is strongly normalizaible if every reducibility sequence starting with t can be extended to a normalization sequence.

Theorem. (See Girard [1972].) Every polymorphic term is strongly normalizable. This is a difficult theorem.

Examples. Polymorphic λ–calculus has such strong expressive power that the usual data types can all be defined in it. For example, integers, lists, trees (see Girard [1988]). In order to define functions it is necessary to define the appropriate types on which they act. Thus the type <u>boolean</u> may be defined as $\Lambda\alpha.\alpha\to(\alpha\to\alpha)$. We can set true $T = \Lambda\alpha.\lambda x^{\alpha}.\lambda y^{\alpha}.x^{\alpha}$ and false $F = \Lambda\alpha.\lambda x^{\alpha}.\lambda y^{\alpha}.y^{\alpha}$ (that is, these are defined as "first" and "second"). It is not hard to check that definition by cases $Dfgb = f$ if $b = T$, $Dfgb = g$ if $b = F$, where f, g have the same type γ and b has type <u>boolean</u>, is defined in the calculus by $Dfgb = (b\gamma)fg$. For then if $b = T$, $b\gamma$ chooses f, the first of the two functions f, g, otherwise g.
Natural numbers can be represented as Church numerals in the style $\underline{n} = \Lambda\alpha.\lambda y^{\alpha\to\alpha}.\lambda x^{\alpha}.y(...(y(x)...)$, with n y's, which term has the type $\underline{\text{integer}} = \Lambda\alpha.\alpha\to((\alpha\to\alpha)\to\alpha)$. Zero is here represented by the term $\Lambda\alpha.\lambda y^{\alpha\to\alpha}.\lambda x^{\alpha}.x$ and successor by $\lambda u^{\underline{\text{integer}}}.\Lambda\alpha.\lambda y^{\alpha\to\alpha}.\lambda x^{\alpha}.y((u\alpha)yx)$. It is also possible to represent integers by $\Lambda\alpha.((\alpha\to\alpha)\to(\alpha\to\alpha))$, in which n is represented by "n–fold iteration". The notion of "polymorphic definable functions of integers" is defined much as in Church's untyped λ–calculus, but not all total recursive functions are polymorphically definable, only the functions of integers provably recursive in second order classical or intuitionistic arithmetic. For more see Girard [1988], Leivant [1983], Reynolds and Plotkin [1989].)

§13. Curry–Howard Isomorphism. The isomorphism will make second order propositions correspond to polymorphic types and will make second order deductions ending in a

proposition correspond to polymorphic terms of the corresponding type. Just as before, term abstraction and term application correspond to implication introduction and implication elimination. What are the new clauses of the definition of the correspondence?

If term t of type A corresponds to a deduction

$$\begin{array}{l} \vdots\, d \\ A \end{array}$$

and σ is a propositional variable not free in the deduction d,
then term $\Lambda\sigma.t$ of type $\Lambda\sigma.A$ corresponds to deduction

$$\frac{\begin{array}{l} \vdots \\ A \end{array}}{\Lambda\sigma\,.\,A}$$

If term t of type $\Lambda\sigma.A$ corresponds to a deduction

$$\begin{array}{l} \vdots\, d \\ \Lambda\sigma\,.\,A \end{array}$$

and B is a legal substitution for σ in A, then the term $\Lambda\sigma.A$ corresponds to

$$\frac{\begin{array}{l} \vdots\, d \\ \Lambda\sigma\,.\,A \end{array}}{A(B/\sigma)}$$

Under the Curry–Howard isomorphism, the reduction of polymorphic terms will correspond perfectly to proof simplification in second order intuitionistic calculus based on $\Lambda, \rightarrow$.

§14. Intuitionistic Zermelo–Fraenkel set theory. IZF is the best developed constructive set theory. IZF attempts to preserve both the expressive power of classical ZF and the mathematician's natural use of higher–order and impredicative concepts. In meeting the requirement of preserving the mathematician's ordinary proof habits, it gets high points

compared to most other constructive foundational systems. One can extract programs from proofs in IZF. This is because IZF has well–behaved recursive realizabilities. We exposit McCarty's [1984]. Nothing in this direction has been implemented so far as we know. We cover this topic to indicate a possible future direction of research. IZF will be a first order theory in intuitionistic logic based on "ϵ" (membership) and "=" (equals). Part of a "Heyting semantics" of IZF is as follows.

– Each set S has as elements names of constructions.
– x is a member of S if x equals a member y of S.
– S equals T means there is a construction leading from each name of a construction $x \epsilon A$ to a name y of an equal construction of B.
– the axioms of IZF should provide for the usual construction of new sets from old, including definitions by transfinite induction, provided that intuitionistic logic is used.

In ZF based on classical logic, definitions by transfinite induction are handled using ordinals, the axiom of foundation, and the replacement axiom. We need to know what axioms to use instead in a constructive context. Classically equivalent axioms are notorious for giving different results in constructive contexts. The "right" definition has to be discovered by analyses of proofs. Here are the axioms of IZF.

Extensionality.
$\forall x \forall y((\forall z)(z \epsilon x \leftrightarrow z \epsilon y) \leftrightarrow x = y)$

Pairing.
$\forall x \forall y \exists z(x \epsilon z \wedge y \epsilon z)$

Union.
$\forall x \exists y \forall z(\forall u \epsilon x\ (z \epsilon u \rightarrow z \epsilon y))$

Separation.
$\forall x \exists y \forall z(z \epsilon y \leftrightarrow (z \epsilon x \wedge \varphi))$

Power.
$\forall x \exists y \forall z((\forall u \epsilon z\ (u \epsilon x) \rightarrow z \epsilon y)$

Infinity.

$\exists x((\exists u \epsilon x\ \forall y\ \neg(y \epsilon u)) \wedge (\forall y \epsilon x\ \exists z \epsilon x\ (y \epsilon z))$

Collection.
$\forall x(\forall y \epsilon x\ \exists z \varphi \longrightarrow \exists u\ \forall y \epsilon x\ \exists z \epsilon u\ \varphi)$

Set induction.
$(\forall x)((\forall y \epsilon x\ \varphi(y) \longrightarrow \varphi(x)) \longrightarrow \forall x\ \varphi$

Notes.
– Using only intuitionistic logic the axiom of choice implies the law of the excluded middle. (Diaconescu, D. Scott), so we do not want it.

– using classical logic the collection axiom is equivalent to the replacement axiom. But Using only intuitionistic logic the collection axiom is stronger than replacement, and we need the additional strength to handle transfinite induction.

– Using classical logic the axiom of set induction is equivalent to the axiom of foundation. But using intuitionistic logic, the axiom of foundation implies the law of the excluded middle, so we do not want it. The axiom of set induction will suffice for the development of ordinals and definitions by transfinite induction, so we adopt it.

– In IZF a set x is an ordinal if x is a transitive sets of transitive sets (Powell). The assumption that every ordinal is either 0, a successor, or a limit, implies the law of the excluded middle, so we do not have this principle in IZF. In IZF arguments about ordinals have to be carried out without separation into these three cases, a common classical technique. These ordinals can be used to provide a notion of rank. Setting $x+1 = x \cup \{x\}$, it can be seen that

$$rk(x) = \bigcup \{rk(x)+1 \mid z \epsilon x\}$$

assigns an ordinal rank to every set in IZF, using the axiom of set induction. For ordinal α the ranked universe of level α is the transitive set

$$V_\alpha = \cup\{P(V_\beta) \mid \beta \epsilon \alpha\}.$$

The axiom of set induction implies that in IZF every set is in some ranked universe V_α. In particular, every element of V_α is a subset of a V_β for $\beta \epsilon \alpha$. Generally, we can

carry out definitions by transinite induction

$$f(\alpha, x) = G(\alpha, x, g)),$$

where g is f restricted to α, provided that there is no case analysis on the ordinals.

McCarty [1984] developed one realizability interpretation for IZF. There are others. His idea was

– to imitate Kleene [1945] realizability, according to which integers n realize propositions (are "evidence for propositions"). So n is regarded as a code for the n th partial recursive function.

– to pass evidence past universal quantifiers, as in Kreisel–Troestra interpretations, see Troelstra [1973]. Thus in the McCarty interpretation the only way to establish a universal is with a single piece of evidence which itself establishes every instance.

– To mimic the definition of V_α with a definition of W_α such that for all members x, S in any W_α, the proposition "$x \in S$" is accompanied by the evidence n for this proposition.

Now S will be a subset of a W_β for a $\beta \in \alpha$, and we always want evidence n for $x \in S$, so we might as well let all such sets S consist of pairs (n, x), where n is an integer and x is an element of W_β, and therefore a subset of a W_γ, $\gamma \in \beta$. But then there has to be a new definition of equality. Equality should mean that the sets of pairs (n, x) have equal elements. That is, S = T should mean that whenever $(n, x) \in S$, there is an m with $(m, x) \in T$ with (m, x) = (n, y), and conversely. This clearly requires a simultaneous inductive definition of membership and equality.

McCarty's Universe W.

$$W_\alpha = \bigcup_{\beta \in \alpha} P(\omega \times W_\beta)$$

$$W = \bigcup_\beta W_\beta$$

This is simply a proper class of names suitable for elementhood evidenced by integers,

which we add as names to a first order language.

Equality.
e realizes $a = b$ iff
$\forall f \forall d([<f, d> \epsilon a \rightarrow (ef)_0$ realizes $d \epsilon b] \wedge [<f, d> \epsilon b \rightarrow (ef)_1$ forces $d \epsilon a])$

Membership.
e realizes $a \epsilon b$ iff
$\exists c(<e_0, c> \epsilon b) \wedge (e_1$ realizes $a = c)$.

Conjunction.
e realizes $\varphi \wedge \psi$ iff
e_0 realizes φ and e_1 realizes ψ.

Disjunction.
e realizes $\varphi \vee \psi$ iff
$(e_0 = 0 \wedge e_1 = \varphi)$ or $(e_0 = 1 \wedge e_1 = \psi)$

Implication.
e realizes $\varphi \rightarrow \psi$ iff
$\forall f$(f realizes φ implies ef is defined and ef realizes ψ).

Existential quantification.
e realizes $\exists x \varphi$ iff
$\exists e$(e realizes $\varphi[x/a]$).

Universal quantification (Kreisel–Troelstra).
e realizes $\forall x \varphi$ iff
$\forall a$(e realizes $\varphi[x/a]$).

We say that φ is true in McCarty's model W if there is an e realizing φ. The semantics is sound for intuitionistic predicate logic with IZF as axioms, and provably so in IZF.

Programs from proofs in IZF.

Here is an application of this realizability to algorithm extraction. Suppose p is a deduction of "$\forall x \epsilon \omega \exists y\ \varphi(x, y) \wedge$ (φ is a recursive predicate)" from the axioms of IZF. From p we can effectively compute a code f of a total recursive function such that $\forall x \epsilon \omega\ \varphi(x, \{f\}(x))$. The meaning is that any proof of $\forall x \exists y \varphi(x, y)$ together with any proof that φ is a recursive predicate can be "compiled" to a program for computing a Skolem function $\{f\}(x)$. In that proof, higher order impredicative notions may be used. The verification of this fact uses:

– the proof of soundness of realisability is uniformly effective in the codes of proofs,
– if Kleene's T(e, x, y) is realized, it is true, metatheoretically.

§15. Resume of extractions of programs from proofs. Automated extraction of programs from proofs in extensions of HA has been an impetus for development of very high level computer languages. Such languages as AUTOMATH (DeBruijn [1973, 1980]), NuPRL (Constable [1986]), the theory of constructions (Huet and Coquand [1985]), and PX (Hayashi [1989]) arise from such logical considerations. This relation between logic and computation is one of the main reasons for studying intuitionistic systems in computer science.

Curry [1958] was the first exploration of the identity of normalization of deductions in logic with normalization of terms in a typed lambda calculus. Prawitz [1965] wrote an important monograph in which normalization in intuitionistic predicate logic is made to look just like λ–calculus normalization. The Curry propositional logic–typed lambda calculus isomorphism was extended to HA by Howard in a widely circulated and influential manuscript in 1969. (Howard used a sequent calculus instead of natural deduction on the logic side, and $\rightarrow$, $\wedge$, $\forall$ on the Heyting arithmetic side.) However, this manuscript was published only in Howard [1980]. Girard read Howard's paper for HA, and in his thesis Girard [1972] extended the Curry–Howard isomorphism first to second order intuitionistic arithmetic HAS and then to full higher order Heyting arithmetic HAH, where unbounded quantification over functions of functions of functions... is allowed. He invented an appropriate typed lambda calculus of higher type (the second order version is known as system F, or polymorphic calculus) in the process. Girard proved strong normalization for these typed lambda calculi, generalizing a method of Tait from proof theory.

Independently Reynolds invented second order polymorphic typed lambda calculus in the 1970's in response to a 1967 program of Strachey. This calculus emphasizes generic or reusable code in a very strong way. In a typed lambda calculus for defining functions from

terms of a type to terms of another type, a program is a lambda term that has to be evaluated when applied to an argument by reduction steps. Reynolds, in order to assure maximum reusability of lambda terms as subprocedures that need not be rewritten, suggested that types should be variable types, and should be specializable to any types without interfering with validity of reduction steps. This entails giving abstraction and application rules for types as well as the usual abstraction and application rules for the terms in which they appear, and strengthens LISP style functional languages significantly. Reynolds and his colleagues validated the reduction procedure appropriate to this calculus, proving a so–called strong normalization in which, unlike untyped lambda calculus, every term reduces to canonical form. Validating this reduction procedure is what is necessary to validate a LISP style interpreter or compiler for such a computer language. But they found that this difficult theorem was a special case of the 1972 thesis of Girard referred to above. This coincidence brought proof theory of intuitionist systems into the mainstream of computer science, where it is today.

Only in the early 1980's was it recognized that the second order Girard lambda calculus, system F, corresponding to HAS, is equivalent to Reynold's polymorphic lambda calculus. This was after those working on Reynold's program had duplicated much of Girard's work of a decade earlier. Girard had showed that under the Curry–Howard isomorphism, strong normalization for logics and for the corresponding lambda calculi were the same. This isomorphism extracts lambda calculus terms (or programs) from proofs in intuitionistic theory. A computer implementation of the strong normalization procedure for the lambda calculus becomes the interpreter or compiler for evaluating terms. Typed lambda terms define functions, and reducing a lambda term applied to an argument to normal form represents computing the value of that function.

What functions does such a formalism compute, are they adequate for a theory of computation? The answer was provided by proof theory. First consider the subject for which Howard invented a term calculus for a fragment of HA, first order Heyting arithmetic. What total functions does his corresponding typed lambda calculus compute? Gödel showed by his functional interpretation that the provably recursive functions of HA are precisely the provably recursive functions of Peano classical first order arithmetic. These were characterized by Gödel as the primitive recursive functionals of finite type, which use ordinary primitive recursion but allow function variables of finite type. This class already contains far more functions than will ever be used in the world of computing.

The corresponding class of functions computable in second order intuitionistic arithmetic HAS, or equivalently in Reynold's polymorphic lambda calculus, is yet wider. Extending Gödel's results, Harvey Friedman [1978] showed that the functions computed in second order intuitionistic logic are exactly those provably recursive in second order classical arithmetic. With a little effort this can be seen to be the class of functions computed by Girard's second order system F, or equivalently Reynold's polymorphic lambda calculus.

There is a provably total recursive function in second order arithmetic enumerating all provably total recursive functions for first order arithmetic, so a diagonal argument shows that second order arithmetic has more provably recursive functions than first order arithmetic. So more functions are computed by the second order polymorphic lambda calculus than by the first order lambda calculus of Howard, which already computes more functions than will ever be needed.

If the typed lambda calculus of Howard corresponding to first order Heyting arithmetic already computes more functions than are needed in practice, why go on to a second order or higher order lambda calculus as Girard's or Reynolds?

– Because of the polymorphism which makes code easily reusable by specializing types.

– Mathematicians naturally write their proofs in second order logic, and their natural datatypes defined by induction are naturally expressed in second order logic. If a proof of the existence of a function G is constructive, the Curry–Howard isomorphism automatically extracts a lambda term which acts as an algorithm (via execution of a strong normalization procedure) to compute G. For this point of view, see Leivant [1983, 1989].

But this feature leads in yet another direction. Martin–Löf was stimulated to invent extremely powerful predicative extensions of HA to transfinite levels. Predicative means roughly that in the construction (or definition) of a mathematical object X, the quantifiers in the definition of X must range only over previously defined objects. This excludes X being introduced by a definition in which the quantifiers range over a domain including X itself. (Limitation to the use of only predicative definitions was advocated by the great French mathematician of the turn of the 20th century, Henri Poincaré.)

Martin–Löf viewed the Curry–Howard isomorphism as a key to giving a computational semantic meaning to logical connectives. The operation on typed lambda terms

corresponding to each rule of deduction for introducing and eliminating a logical connective assigns a computational mea ing to that connective. The Martin–Löf systems are built by a simultaneous inductive definition of both the logic and a corresponding generalization of typed lambda calculus. He does not distinguish between the two sides of the Curry–Howard isomorphism, but this is because his is indeed a simultaneous inductive definition of each in terms of both. The predicative character of the system is due to its simultaneous inductive definition of logical deduction and term reduction. His systems are described in a formalization due to Aczel in Beeson [1985]. A Martin–Löf system was chosen by Constable in the middle 1970's as the best candidate for implementation of a language to extract programs from constructive proofs. This language is now embedded in Constable's NuPRL. The AUTOMATH language of DeBruijn is similar but was developed independently of formal knowledge of the Curry–Howard isomorphism. Similar languages based on other primitives are due to Feferman (explicit mathematics) and Aczel (Frege Structures). The Huet–Coquand theory of constructions [1985] is a powerful impredicative extension of Girard's ω–level F^{ω} being implemented within ULYSSES at Odyssey Research.

Every notion of recursive realizability extracts programs from proofs. Realizability for intuitionistic analysis and set theory have been investigated by Kleene, Kreisel and Troelstra, and McCarty. These realizabilities have not yet been exploited in computer science, but have some promise since they extract programs from the mathematician's ordinary proofs, so long as they are constructive, without having to avoid higher order or transfinite notions.

BIBLIOGRAPHY

Starred entries (*) contain extensive bibliographies.

Part I: General References

M. J. Beeson(*) [1985], *Foundations of Constructive Mathematics*, Springer–Verlag, Berlin.
An encyclopedic reference summarizing the metamathematical work of the field in the past twenty years. The exposition is based on partial applicative structures as models for realizability. This treatise explains many of the constructive formal systems in use in computer science, such as Martin–Löf's, as well as realizability and forcing. Eschews the Kripke frame approach.

Barendregt, H. [1984], *The Lambda Calculus: Its Syntax and Semantics* revised edition, North Holland, Amsterdam.

D. van Dalen [1983], *Logic and Structure*, second ed., Springer–Verlag, Berlin.
An undergraduate logic text with an excellent chapter on intuitionistic logic and frames.

D. van Dalen(*) [1986], "Intuitionistic logic", in *The Handbook of Philosophical*

Logic, vol. III, 225–339, D. Reidel, Dordrecht.
The best succinct introduction yet written.

A. G. Dragalin(*) [1987], *Mathematical Intuitionism: Introduction to Proof Theory*, Translations of Mathematical Monographs 67, AMS, Providence, R. I.
Assumes no background and gives some unity to Kripke frames and Kleene's realizability semantics.

J–Y. Girard [1987], *Proof Theory and Logical Complexity*, vol. I, Studies in Proof Theory, Bibliopolis, Naples. The clearest version of proof theory yet. A great emphasis on mathematuical structure.

J–Y. Girard, Y. Lafont, and P. Taylor [1989], *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press.
The first legible exposition of strong normalization for system F (or polymorphic calculus) in textbook form.

S. C. Kleene [1952], *Introduction to Metamathematics*, North–Holland (1971 edition), Amsterdam.
This classic text still has the best exposition of the relation between classical and intuitionist logic.

G. Odifreddi [1989], *Classical Recursion Theory*, North Holland.

D. Prawitz [1965], *Natural Deduction*, Almqvist and Wiskell, Stockholm.
This exposition of natural deduction made the isomorphism of natural deduction calculi with suitable typed lambda calculi transparent.

A. S. Troelstra(*) [1973], *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, Mathematical Lecture Notes 344, Springer–Verlag, Berlin.
The encyclopedic reference to the subject. Includes a book–length chapter on Kripke frames, by Smorynski, and chapters on almost every other aspect of the subject known at the time.

A. Troelstra and D. van Dalen(*) [1988], *Constructivism in Mathematics, An Introduction*, North–Holland, vol. 1, 2.
The most complete survey, incorporating much of the work of the 1980's.

Other References

L. Brownston, R. Farrell, E. Kant, and N. Martin [1985], *Programming expert systems in OPS5*, Addison–Wesley, Reading, Mass.

N. De Bruijn [1973], *Automath: a Language for Mathematics*, Les Presses de l'Universite de Montreal, Montreal.

N. De Bruijn [1980], "A survey of the project AUTOMATH", in *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 579–606.

Constable, R. et al. [1986], *Implementing Mathematics in the NuPrl System*, Prentice–Hall, Englewood Cliffs, N. J.

Coquand, T. [1985], "Une Theorié des Constructions", Thèse de Troisième cycle, Universitè Paris VII.

Coquand, T., Huet, G. [1985], "Constructions: a higher order proof system for mechanizing mathematics", in Proc. EUROCAL '85, Linz, Austria, Lecture notes in Computer Science vol. 203, Springer–Verlag, Berlin.

Coquand, T., Huet, G. [198?], "A selected bibliography on constructive mathematics, intuitionistic type theory, and higher order deduction", preprint..

Curry, H. and Feys, R. [1958], *Combinatory Logic*, North Holland, Amsterdam.

M. Fitting [1983], *Proof Methods for Modal and Intuitionistic Logics*, D. Reidel. Dordrecht.

H. Friedman [1978], "Classically and intuitionistically provable recursive functions", in Higher Set Theory (Oberwolhfach 1977), Lecture Notes in Mathematics No. 669, Springer–Verlag, Berlin, 1978, 21–27.

M. Fourman and D. Scott [1977], "Sheaves and logic", in *Applications of Sheaves*, (Fourman, Mulvey and Scott, eds.), Mathematical Lecture Notes 753, Springer–Verlag,

Berlin, 302–401.

G. Gentzen [1969], *The Collected papers of Gerhard Gentzen*, E. Szabo, ed., North Holland, Amsterdam.

J–Y. Girard [1971], "Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types", in *Proceedings of the Second Scandinavian Logic Symposium*, (J. E. Fenstad, ed.), North–Holland, Amsterdam.

J–Y. Girard [1972], "Interprétation fonctionelle et élimination de coupures de l'arithmétique d'ordre supérieur," Thèse de Doctorat d'état, Université Paris VII.

J–Y. Girard [1986], "The system of variable types, fifteen years later", Theoretical Computer Science 45, 159–192.

J–Y. Girard [1987], "Lambda calcul typé", notes for his 1986–87 course at Université Paris VII.

K. Gödel [1958], "Uber eine bisher noch benutze Erweiterung des finiten standpunktes", Dialectica 12, 280–287.

N. Goodman [1970], "A theory of constructions equivalent to arithmetic", in Myhill, Kino, Vesley, Intuitionism and Proof Theory, North Holland, Amsterdam.

N. Goodman [1973], "The arithmetic theory of constructions", Cambridge Summer School in Mathematics and Logic (Mathias and Rogers, eds.), Springer–Verlag, Berlin.

Hayashi [1989], "PX", Proceedings of the Workshop on the Logical Foundations of Programming Languages, Addison–Wesley, N. Y.

A. Heyting [1930], "Die formalen Regeln der intuitionistischen Logik" I, II, III, Sitzungberichte der Preussischen Akademie von Wissenschaften. Physikalisch–mathematische Klasse, 42–56, 57–91, 158–169.

A. Heyting [1934], *Mathematische grundlagenforschung Intuitionismus Beweistheorie*, Springer–Verlag, Berlin.

A. Heyting [1955], *Les Fondements des Mathematiques Intuitionnisime Theorie de la Demonstration*, Gauthier–Villars, Paris:

J. Hindley and J. Seldin [1986], *Introduction to Combinators and the λ–calculus*, London Mathematical Society Student Texts I, Cambridge University Press.

J. Hopcroft and J. D. Ullman [1979], *Introduction to Automata Theory, Languages and Computation*, Addison–Wesley, Reading, Massachusetts.

W. A. Howard [1980], "The Formulae–as–types notion of construction", in *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, Academic Press. 479–490.

G. Huet [1989], "Topics in constructive set theory: An introduction", in Proceedings from the Workshop in Functional Programming Languages, Austin, Texas Addison–Wesley, Reading, Mass, to appear.

S. C. Kleene [1945], "On the interpretation of intuitionistic number theory", J.S.L. 10, 109–124.

S. C. Kleene [1960], "Realizability and Shanin's algorithm for the constructive deciphering of mathematical sentences", Logique et Analyse, 154–165.

S. C. Kleene and R. Vesley [1965], *The Foundations of Intuitionistic Mathematics*. North–Holland, Amsterdam.

S. C. Kleene [1973], "Realizability: a retrospective survey", in Cambridge Summer School in Mathematical Logic (Mathias. and Rogers, eds.), Springer–Verlag, 1969.

J. Lambek, "From λ–calculus to Cartesian closed categories", in *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 375–402.

J. Lambek and P. J. Scott [1986], *Introduction to higher order categorical logic*, Cambridge Studies in Advanced Mathematics 7, Cambridge.

H. Läuchli [1970], "An abstract notion of realizability for which predicate calculus is complete," in *Intuitionism and Proof Theory*, (Kino, Myhill, Vesley, eds.), North–Holland, Amsterdam.

D. Leivant [1983], "Polymorphic type inference", in *Proc. 10th ACM Symposium on the Principles of Programming Languages*, ACM, New York, 88–98.

D. Leivant [1983a], "Reasoning about functional programs and complexity classes associated with type disciplines," in *24th Annual Symposium on Foundations of Computer Science*, 460–496.

D. Leivant [199?], *Intuitionistic Formal Systems*, to appear.

P. Martin–Löf [1980], "Constructive mathematics and computer programming", Logic, Methodology, and Philosophy of Science 6, North Holland, Amsterdam.

P. Martin–Löf [1984], *Intuitionistic Type Theory*, Studies in Proof Theory, Bibliopolos.

J. Lloyd, *Foundations of Logic Programming*, Springer–Verlag, 1987.

McCarty,C. D. [1984], *Realizability and Recursive Mathematics*, Ph. D. thesis, Oxford University.

Meseguer, J. [1988], "Relating Models of Polymorphism", SRI International Technical Report.

A. Nerode [1989?], "Some Lectures on Intuitionistic Logic I", CIME Montecatini Volume, Springer–Verlag Lecture Notes, to appear.

G. Pottinger [1977], "Normalization as a homomorphi. image of cut–elimination", Ann. Math. Logic 12, 327–357.

H. Rasiowa and R. Sikorski [1963], *Mathematics of Metamathematics*, Monographie Matematyczne, t. 41, Warsawa.

J. Reynolds [1974], "Towards a Theory of Type Structure", in Proceedings, Colloque sur la Programmation, Springer Verlag, Berlin, 408–425.

J. Reynolds [1974], "Polymorphism is not set–theoretic", in Kahn, MacQueen, Plotkin, eds., Semantics of Data Types, Springer–Verlag, Berlin.

J. Reynolds and G. Plotkin [1989], "On functors expressible in the polymorphic typed lambda calculus", Proceedings of the Workshop on the Logical Foundations of Programming Languages, Addison–Wesley, N. Y.

J. Robinson [1965], "A machine oriented logic based on the resolution principle", JACM 12, 23–41.

D. Scott [1980], "Relating theories of the lambda calculus", in *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 403–450.

S. Stenlund [1972], *Combinators, λ–terms and proof theory*, D. Reidel, Dordrecht.

D. Turner [1985], "Miranda: a non–strict functional language with polymorphic types", in Functional Programming Languages and Comnputer Architecture, J. P. Jouanneaud, ed., Lecture Notes in Computer Science, vol. 201, Springer–Verlag, Berlin.

J. Zucker [1974], "Cut–elimination and normalization", Ann. Math. Logic, 1–112.

See the LICS (Logic in Computer Science) annual conference volumes for references to current computer science applications.

* Research supported in part by ARO Grant DAAG–29–85–C–0018 and NSF grant MCS–8301850

** We are grateful to Prof. J. N. Crossley and to Sloan Fellow James Lipton for substantial suggestions.